**Industry Assignment**

**Name: Ritesh Prakashrao Gandalwad**

**Roll No: 13-A**

**Subject: Data Structure And Algorithm**

**Assignment Title:** "Write a program to implement a word suggestion system for crossword puzzles using a Trie data structure. The program should allow the user to input the word length, number of known characters, and a partially completed word (with placeholders for unknown letters). The system should then provide a list of possible word matches from a predefined dictionary, utilizing a Trie data structure for efficient matching."

**Problem Statement:** This assignment involves using the Trie data structure, commonly used in various real-world applications like autocomplete systems, search engines, and word games. The task is to implement a word suggestion system for crossword puzzles, allowing the user to input partial words with missing letters (represented as '?') and receive possible word suggestions based on the given pattern. The system could also leverage DFS to efficiently explore potential word matches in a predefined dictionary.

**Description**

1. **User Input**:
   * **Word length**: Specifies the length of the target word.
   * **Number of known letters**: Indicates how many letters in the word are known.
   * **Pattern**: Partially completed word with ? as a placeholder for unknown letters (e.g., f??st).
2. **Processing**:
   * The program uses a **Trie** to store and search for words from a dictionary.
   * Suggests words that match the given pattern and satisfy the number of known letters.
3. **Output**:
   * A list of matching words from the dictionary.  
     Example: Input f??st with word length 5 and 3 known letters might output:  
     ['frost', 'foist', 'first', 'feast', 'feist', 'faust'].

C Code:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#define ALPHABET\_SIZE 26

#define MAX\_WORD\_LENGTH 100

#define MAX\_RESULTS 100

// Trie node structure

typedef struct TrieNode {

struct TrieNode \*children[ALPHABET\_SIZE];

bool isEndOfWord;

} TrieNode;

// Create a new Trie node

TrieNode \*createNode() {

TrieNode \*node = (TrieNode \*)malloc(sizeof(TrieNode));

node->isEndOfWord = false;

for (int i = 0; i < ALPHABET\_SIZE; i++) {

node->children[i] = NULL;

}

return node;

}

// Insert a word into the Trie

void insert(TrieNode \*root, const char \*word) {

TrieNode \*node = root;

while (\*word) {

int index = \*word - 'a';

if (node->children[index] == NULL) {

node->children[index] = createNode();

}

node = node->children[index];

word++;

}

node->isEndOfWord = true;

}

// Recursive function to search for matching words

void searchPattern(TrieNode \*node, const char \*pattern, int index, char \*currentWord, int currentLength, char results[][MAX\_WORD\_LENGTH], int \*resultCount) {

if (node == NULL) {

return;

}

if (pattern[index] == '\0') {

if (node->isEndOfWord) {

currentWord[currentLength] = '\0';

strcpy(results[\*resultCount], currentWord);

(\*resultCount)++;

}

return;

}

if (pattern[index] == '?') {

for (int i = 0; i < ALPHABET\_SIZE; i++) {

if (node->children[i]) {

currentWord[currentLength] = 'a' + i;

searchPattern(node->children[i], pattern, index + 1, currentWord, currentLength + 1, results, resultCount);

}

}

} else {

int charIndex = pattern[index] - 'a';

if (node->children[charIndex]) {

currentWord[currentLength] = pattern[index];

searchPattern(node->children[charIndex], pattern, index + 1, currentWord, currentLength + 1, results, resultCount);

}

}

}

// Find matching words

void findMatches(TrieNode \*root, const char \*pattern, char results[][MAX\_WORD\_LENGTH], int \*resultCount) {

char currentWord[MAX\_WORD\_LENGTH];

\*resultCount = 0;

searchPattern(root, pattern, 0, currentWord, 0, results, resultCount);

}

// Load dictionary

void loadDictionary(TrieNode \*root) {

const char \*dictionary[] = {"frost", "foist", "first", "feast", "feist", "faust", "toast", "ghost", "coast"};

int dictionarySize = sizeof(dictionary) / sizeof(dictionary[0]);

for (int i = 0; i < dictionarySize; i++) {

insert(root, dictionary[i]);

}

}

int main() {

TrieNode \*root = createNode();

// Load dictionary into the Trie

loadDictionary(root);

// User inputs

int wordLength, knownLettersCount;

char pattern[MAX\_WORD\_LENGTH];

printf("Enter the word length: ");

scanf("%d", &wordLength);

printf("Enter the number of known letters: ");

scanf("%d", &knownLettersCount);

printf("Enter the pattern (use '?' for unknown letters): ");

scanf("%s", pattern);

// Validate inputs

if (strlen(pattern) != wordLength) {

printf("Error: Pattern length must match the word length.\n");

return 1;

}

int unknownCount = 0;

for (int i = 0; i < wordLength; i++) {

if (pattern[i] == '?') {

unknownCount++;

}

}

if (unknownCount != wordLength - knownLettersCount) {

printf("Error: Number of unknown letters does not match the input.\n");

return 1;

}

// Find matches

char results[MAX\_RESULTS][MAX\_WORD\_LENGTH];

int resultCount;

findMatches(root, pattern, results, &resultCount);

if (resultCount > 0) {

printf("Possible matches:\n");

for (int i = 0; i < resultCount; i++) {

printf("%s\n", results[i]);

}

} else {

printf("No matches found.\n");

}

return 0;

}

Output:

